Introduction to I/O

CSE 421/521: Operating Systems
Karthik Dantu

Slides adopted from CS162 class at Berkeley, CSE 451 at U-Washington and CSE 421 by Prof Kosar at UB
Assignment #1 - Clarifications

• Grading
  – 5% - Design
  – 10% - Implementation/tests

• Git
  – Working with github to set up repos
  – Should be up by early next week

• Details of scheduling
  – In coming classes and recitations
  – Assignment given early

• Deadlines
  – 9/26 – Design document
  – 10/3 – Implementation via Autograder
# Recall: UNIX System Structure

## User Mode
- **Applications** (the users)
- **Standard Libs**
  - shells and commands
  - compilers and interpreters
  - system libraries

## Kernel Mode
- **System-call interface to the kernel**
  - signals terminal handling
  - character I/O system
  - terminal drivers
  - file system
  - swapping block I/O system
  - disk and tape drivers
  - CPU scheduling
  - page replacement
  - demand paging
  - virtual memory

## Hardware
- **Kernel interface to the hardware**
  - terminal controllers
  - terminals
  - device controllers
  - disks and tapes
  - memory controllers
  - physical memory
How Does the Kernel Provide Services?

• You said that applications request services from the operating system via syscall, but ...

• I’ve been writing all sort of useful applications and I never ever saw a “syscall” !!!

• That’s right.

• It was buried in the programming language runtime library (e.g., libc.a)

• ... Layering
OS Run-Time Library

Proc 1   Proc 2   Proc n

Appln   login   Window Manager

OS library   OS library   OS library

OS
A Kind of Narrow Waist

Compilers

Word Processing

Web Browsers

Email

Web Servers

Databases

Portable OS Library

System Call Interface

Portable OS Kernel

Platform support, Device Drivers

User

System

Software

x86

PowerPC

ARM

Hardware

Ethernet (1Gbs/10Gbs) 802.11 a/g/n/ac SCSI Graphics Thunderbolt

Application / Service

Operating System (OS)
Key Unix I/O Design Concepts

• Uniformity
  – file operations, device I/O, and interprocess communication through `open`, `read/write`, `close`
  – Allows simple composition of programs
    • `find | grep | wc` ...

• Open before use
  – Provides opportunity for access control and arbitration
  – Sets up the underlying machinery, i.e., data structures

• Byte-oriented
  – Even if blocks are transferred, addressing is in bytes

• Kernel buffered reads
  – Streaming and block devices looks the same
  – read blocks process, yielding processor to other task
Putting it together: web server

1. network socket read
2. copy arriving packet (DMA)
3. kernel copy
4. parse request
5. file read
6. disk request
7. disk data (DMA)
8. kernel copy
9. format reply
10. network socket write
11. kernel copy from user buffer to network buffer
12. format outgoing packet and DMA

Kernel buffer reads
Key Unix I/O Design Concepts

• Uniformity
  – file operations, device I/O, and interprocess communication through `open`, `read/write`, `close`
  – Allows simple composition of programs
    • `find | grep | wc` ...

• Open before use
  – Provides opportunity for access control and arbitration
  – Sets up the underlying machinery, i.e., data structures

• Byte-oriented
  – Even if blocks are transferred, addressing is in bytes

• Kernel buffered reads
  – Streaming and block devices looks the same
  – read blocks process, yielding processor to other task

• Kernel buffered writes
  – Completion of out-going transfer decoupled from the application, allowing it to continue
Putting it together: web server

1. network socket read
2. copy arriving packet (DMA)
3. kernel copy
4. parse request
5. file read
6. disk request
7. disk data (DMA)
8. kernel copy
9. format reply
10. network socket write
11. kernel copy from user buffer to network buffer
12. format outgoing packet and DMA

Kernel buffer writes

Hardware
Network interface
Disk interface

Request
Reply
Key Unix I/O Design Concepts

• Uniformity
  – file operations, device I/O, and interprocess communication through open, read/write, close
  – Allows simple composition of programs
    • find | grep | wc ...

• Open before use
  – Provides opportunity for access control and arbitration
  – Sets up the underlying machinery, i.e., data structures

• Byte-oriented
  – Even if blocks are transferred, addressing is in bytes

• Kernel buffered reads
  – Streaming and block devices looks the same
  – read blocks process, yielding processor to other task

• Kernel buffered writes
  – Completion of out-going transfer decoupled from the application, allowing it to continue

• Explicit close
I/O & Storage Layers

- Application / Service
  - High Level I/O
  - Low Level I/O
    - Syscall
  - File System
  - I/O Driver

- Commands and Data Transfers
  - Disks, Flash, Controllers, DMA

- Streams
- Handles
- Registers
- Descriptors
The File System Abstraction

• High-level idea
  – Files live in hierarchical namespace of filenames

• File
  – Named collection of data in a file system
  – File data
    • Text, binary, linearized objects
  – File Metadata: information about the file
    • Size, Modification Time, Owner, Security info
    • Basis for access control

• Directory
  – “Folder” containing files & Directories
  – Hierarchical (graphical) naming
    • Path through the directory graph
    • Uniquely identifies a file or directory
      – /home/ff/cs162/public_html/fa17/index.html
  – Links and Volumes (later)
C High-Level File API – Streams (review)

- Operate on “streams” - sequence of bytes, whether text or data, with a position

```c
#include <stdio.h>
FILE *fopen( const char *filename, const char *mode);
int fclose( FILE *fp );
```

<table>
<thead>
<tr>
<th>Mode Text</th>
<th>Binary</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>rb</td>
<td>Open existing file for reading</td>
</tr>
<tr>
<td>w</td>
<td>wb</td>
<td>Open for writing; created if does not exist</td>
</tr>
<tr>
<td>a</td>
<td>ab</td>
<td>Open for appending; created if does not exist</td>
</tr>
<tr>
<td>r+</td>
<td>rb+</td>
<td>Open existing file for reading &amp; writing.</td>
</tr>
<tr>
<td>w+</td>
<td>wb+</td>
<td>Open for reading &amp; writing; truncated to zero if exists, create otherwise</td>
</tr>
<tr>
<td>a+</td>
<td>ab+</td>
<td>Open for reading &amp; writing. Created if does not exist. Read from beginning, write as append</td>
</tr>
</tbody>
</table>

*Don’t forget to flush*
Connecting Processes, Filesystem, and Users

• Process has a ‘current working directory’

• Absolute Paths
  – /home/kdantu/cs421

• Relative paths
  – index.html, ./index.html - current WD
  – ../index.html - parent of current WD
  – ~, ~kdantu - home directory
C API Standard Streams

• Three predefined streams are opened implicitly when a program is executed
  – FILE *stdin – normal source of input, can be redirected
  – FILE *stdout – normal source of output, can be redirected
  – FILE *stderr – diagnostics and errors, can be redirected

• STDIN / STDOUT enable composition in Unix
  – Recall: Use of pipe symbols connects STDOUT and STDIN
    • find | grep | wc ...

•
C high level File API – Stream Ops

#include <stdio.h>

// character oriented
int fputc( int c, FILE *fp ); // rtn c or EOF on err
int fputs( const char *s, FILE *fp ); // rtn >0 or EOF

int fgetc( FILE * fp );
char *fgets( char *buf, int n, FILE *fp );

DESCRIPTION

The fgets() function reads at most one less than the number of characters specified by size from the given stream and stores them in the string str. Reading stops when a newline character is found, at end-of-file or error. The newline, if any, is retained. If any characters are read and there is no error, a ‘\0’ character is appended to end the string.
C high level File API – Stream Ops

#include <stdio.h>

// character oriented
int fputc( int c, FILE *fp ); // rtn c or EOF on err
int fputs( const char *s, FILE *fp ); // rtn >0 or EOF

int fgetc( FILE * fp );
char *fgets( char *buf, int n, FILE *fp );

// block oriented
size_t fread(void *ptr, size_t size_of_elements,
             size_t number_of_elements, FILE *a_file);

size_t fwrite(const void *ptr, size_t size_of_elements,
              size_t number_of_elements, FILE *a_file);
#include <stdio.h>

// character oriented
int fputc( int c, FILE *fp ); // rtn c or EOF on err
int fputs( const char *s, FILE *fp ); // rtn >0 or EOF
int fgetc( FILE *fp );
char *fgets( char *buf, int n, FILE *fp );

// block oriented
size_t fread(void *ptr, size_t size_of_elements,
               size_t number_of_elements, FILE *a_file);
size_t fwrite(const void *ptr, size_t size_of_elements,
              size_t number_of_elements, FILE *a_file);

// formatted
int fprintf(FILE *restrict stream, const char *restrict format,
           ...);
int fscanf(FILE *restrict stream, const char *restrict format,
           ...);
Example Code

```c
#include <stdio.h>

#define BUFLEN 256
FILE *outfile;
char mybuf[BUFLEN];

int storetofile() {
    char *instring;

    outfile = fopen("/usr/homes/testing/tokens", "w+"); 
    if (!outfile) 
        return (-1);  // Error!
    while (1) {
        instring = fgets(mybuf, BUFLEN, stdin);  // catches overrun!

        // Check for error or end of file (^D)
        if (!instring || strlen(instring)==0) break;

        // Write string to output file, exit on error
        if (fputs(instring, outfile)< 0) break;
    }
    fclose(outfile);  // Flushes from userspace
}
```
C Stream API positioning

int fseek(FILE *stream, long int offset, int whence);
long int ftell (FILE *stream)
void rewind (FILE *stream)

- Preserves high level abstraction of uniform stream of objects
- Adds buffering for performance
What’s below the surface??

Application / Service

High Level I/O

Low Level I/O

Syscall

File System

I/O Driver

streams

handles

registers

descriptors

Commands and Data Transfers

Disks, Flash, Controllers, DMA
C Low level I/O

- Operations on File Descriptors – as OS object representing the state of a file
  - User has a “handle” on the descriptor

```c
#include <fcntl.h>
#include <unistd.h>
#include <sys/types.h>

int open (const char *filename, int flags [, mode_t mode])
int creat (const char *filename, mode_t mode)
int close (int filedes)
```

Bit vector of:
- Access modes (Rd, Wr, …)
- Open Flags (Create, …)
- Operating modes (Appends, …)

Bit vector of Permission Bits:
- User|Group|Other X R|W|X

#include <unistd.h>

STDIN_FILENO - macro has value 0
STDOUT_FILENO - macro has value 1
STDERR_FILENO - macro has value 2

int fileno (FILE *stream)

FILE * fdopen (int filedes, const char *opentype)

• Crossing levels: File descriptors vs. streams
• Don’t mix them!
C Low Level Operations

ssize_t read (int filedes, void *buffer, size_t maxsize)
- returns bytes read, 0 => EOF, -1 => error

ssize_t write (int filedes, const void *buffer, size_t size)
- returns bytes written

off_t lseek (int filedes, off_t offset, int whence)

int fsync (int fildes) – wait for i/o to finish
void sync (void) – wait for ALL to finish

• When write returns, data is on its way to disk and can be read, but it may not actually be permanent!
And lots more!

- TTYs versus files
- Memory mapped files
- File Locking
- Asynchronous I/O
- Generic I/O Control Operations
- Duplicating descriptors
  
  ```c
  int dup2 (int old, int new)
  int dup (int old)
  ```
Another example: lowio-std.c

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>

#define BUFSIZE 1024

int main(int argc, char *argv[])
{
    char buf[BUFSIZE];
    ssize_t writelen = write(STDOUT_FILENO, "I am a process.\n", 16);

    ssize_t readlen = read(STDIN_FILENO, buf, BUFSIZE);

    ssize_t strlen = snprintf(buf, BUFSIZE,"Got %zd chars\n", readlen);

    writelen = strlen < BUFSIZE ? strlen : BUFSIZE;
    write(STDOUT_FILENO, buf, writelen);

    exit(0);
}
What’s below the surface??

Application / Service

High Level I/O

Low Level I/O

Syscall

File System

I/O Driver

Streams

Handles

Registers

Descriptors

Commands and Data Transfers

Disks, Flash, Controllers, DMA
Recall: SYSCALL

- Low level lib parameters are set up in registers and syscall instruction is issued
  - A type of synchronous exception that enters well-defined entry points into kernel
What’s below the surface??

File descriptor number - an int

File Descriptors - a struct with all the info about the files

Application / Service
- streams
- handles
- registers
- descriptors

High Level I/O

Low Level I/O
- Syscall

File System

I/O Driver

Commands and Data Transfers
- Disks, Flash, Controllers, DMA
Internal OS File Descriptor

• Internal Data Structure describing everything about the file
  – Where it resides
  – Its status
  – How to access it

• Pointer:
  ```c
  struct file *file
  ```
File System: from syscall to driver

In fs/read_write.c

```c
ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t *pos) {
    ssize_t ret;
    if (!(file->f_mode & FMODE_READ)) return -EBADF;
    if (!file->f_op || (!file->f_op->read && !file->f_op->aio_read))
        return -EINVAL;
    if (unlikely(!access_ok(VERIFY_WRITE, buf, count))) return -EFAULT;
    ret = rw_verify_area(READ, file, pos, count);
    if (ret >= 0) {
        count = ret;
        if (file->f_op->read)
            ret = file->f_op->read(file, buf, count, pos);
        else
            ret = do_sync_read(file, buf, count, pos);
        if (ret > 0) {
            fsnotify_access(file->f_path.dentry);
            add_rchar(current, ret);
        }
        inc_syscr(current);
    }
    return ret;
}
```
Lower Level Driver

• Associated with particular hardware device
• Registers / Unregisters itself with the kernel
• Handler functions for each of the file operations

```c
struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *
    ssize_t (*aio_read) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
    ssize_t (*aio_write) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
    int (*readdir) (struct file *, void *, filldir_t);
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
    int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *, f1_owner_t id);
    int (*release) (struct inode *, struct file *);
    int (*fsync) (struct file *, struct dentry *, int datasync);
    int (*fasync) (int, struct file *, int);
    int (*flock) (struct file *, int, struct file_lock *);
    [...]
};
```
Recall: Device Drivers

- **Device Driver**: Device-specific code in the kernel that interacts directly with the device hardware
  - Supports a standard, internal interface
  - Same kernel I/O system can interact easily with different device drivers
  - Special device-specific configuration supported with the `ioctl()` system call

- Device Drivers typically divided into two pieces:
  - Top half: accessed in call path from system calls
    - implements a set of **standard, cross-device calls** like `open()`, `close()`, `read()`, `write()`, `ioctl()`, `strategy()`
    - This is the kernel’s interface to the device driver
    - Top half will **start** I/O to device, may put thread to sleep until finished
  - Bottom half: run as interrupt routine
    - Gets input or transfers next block of output
    - May wake sleeping threads if I/O now complete
Life Cycle of An I/O Request

User Program
- request I/O
- system call
- can already satisfy request?
  - yes
  - transfer data (if appropriate) to process, return completion or error code
  - return from system call
  - no
  - send request to device driver, block process if appropriate

Kernel I/O Subsystem
- device-driver commands
- receive interrupt, store data in device-driver buffer if input, signal to unblock device driver
- I/O completed, generate interrupt

Device Driver Top Half
- process request, issue commands to controller, configure controller to block until interrupted
- device-controller commands
- interrupt handler
- interrupt

Device Driver Bottom Half
- device controller
- monitor device, interrupt when I/O completed

Device Hardware
- I/O completed, indicate state change to I/O subsystem
So what happens when you `fgetc`?

Application / Service

High Level I/O

Low Level I/O

Syscall

File System

I/O Driver

streams
handles
registers
descriptors
Commands and Data Transfers
Disks, Flash, Controllers, DMA
Communication between processes

• Can we view files as communication channels?

\[ \text{write}(wfd, wbuf, wlen); \]

\[ n = \text{read}(rfd, rbuf, rmax); \]

• Producer and Consumer of a file may be distinct processes
  – May be separated in time (or not)

• However, what if data written once and consumed once?
  – Don’t we want something more like a queue?
  – Can still look like File I/O!
Communication Across the world looks like file IO

```c
write(wfd, wbuf, wlen);
```

```c
n = read(rfd, rbuf, rmax);
```

- Connected queues over the Internet
  - But what’s the analog of open?
  - What is the namespace?
  - How are they connected in time?
Request Response Protocol

Client (issues requests)  
write(rqfd, rqbuf, buflen);

Server (performs operations)  
wait

n = read(rfd, rbuf, rmax);

service request

write(wfd, respbuf, len);

n = read(resfd, resbuf, resmax);
Request Response Protocol

Client (issues requests)

write(rqfd, rqbuf, buflen);

requests

n = read(rfd, rbuf, rmax);

service request

wait

responses

Server (performs operations)

n = read(resfd, resbuf, resmax);

write(wfd, respbuf, len);
Client-Server Models

- File servers, web, FTP, Databases, ...
- Many clients accessing a common server
Conclusion (I)

- System Call Interface is “narrow waist” between user programs and kernel

- Streaming IO: modeled as a stream of bytes
  - Most streaming I/O functions start with “f” (like “fread”)
  - Data buffered automatically by C-library functions

- Low-level I/O:
  - File descriptors are integers
  - Low-level I/O supported directly at system call level

- STDIN / STDOUT enable composition in Unix
  - Use of pipe symbols connects STDOUT and STDIN
    - `find | grep | wc ...`
Conclusion (II)

• Device Driver: Device-specific code in the kernel that interacts directly with the device hardware
  – Supports a standard, internal interface
  – Same kernel I/O system can interact easily with different device drivers

• File abstraction works for inter-processes communication (local or Internet)